IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of:	)	
	)	
Jeffrey A. BEDELL, et al.	)	
	)	
Serial No.: 09/883,302	)	Art Unit: 2152
	)	
Filed: June 19, 2001	)	Examiner: H. Alaubaidi

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF PRIOR INVENTION UNDER 37 C.F.R. § 1.131

Sir:

We, Jeffrey A. Bedell, Yinong Chen, Benjamin Z. Li, Fabrice Martin, Sadanand Sahasrabudhe, and Jun Yuan, hereby declare that we are co-inventors of the invention that is claimed in the above-identified patent application. Prior to April 21, 2001, we conceived of and reduced to practice the invention that is claimed in the above-identified patent application.

Internal documentation related to the development of this feature demonstrates conception as early as Feb. 15, 1999 with diligence through its actual reduction to practice on or after June 20, 2000. Exhibit A is a February 15, 1999 document entitled "Zen and the Art of VLDB Syntax Generation" that describes one embodiment of using syntax patterns that are selected, populated and then run against a database. Exhibit B dated 4/20/99 through 10/14/99 is entitled "CreateTable VSG" and demonstrates diligence in the development of the invention through the end of 1999. The product went through several rounds of tests, a confidential beta release and then final release on or before June of 2000 in a product called MicroStrategy 7.0.

We further hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so

made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DECLARANT: _____
Jeffrey A. Bedell

Date: 6/16/2005

DECLARANT: _____
Yinong Chen

Date: 6/16/2005

DECLARANT: _____
Benjamin Z. Li

Date: 6/16/2005

DECLARANT: _____
Fabrice Martin

Date: _____

DECLARANT: _____
Sadanand Sahasrabudhe

Date: _____

DECLARANT: _____
Jun Yuan

Date: 06/16/2005

made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DECLARANT: _____
Jeffrey A. Bedell

Date: _____

DECLARANT: _____
Yinong Chen

Date: _____

DECLARANT: _____
Benjamin Z. Li

Date: _____

DECLARANT: _____
Fabrice Martin

Date: 6/20/2005

DECLARANT: _____
Sadanand Sahasrabudhe

Date: _____

DECLARANT: _____
Jun Yuan

Date: _____

made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DECLARANT: _____
Jeffrey A. Bedell

Date: _____

DECLARANT: _____
Yinong Chen

Date: _____

DECLARANT: _____
Benjamin Z. Li

Date: _____

DECLARANT: _____
Fabrice Martin

Date: _____

DECLARANT: _____
Sadanand Sahasrabudhe

Date: 6/16/2005

DECLARANT: _____
Jun Yuan

Date: _____

Zen and the Art of VLDB Syntax Generation

ABSTRACT

Type your abstract or summary here.
Both Abstract and History are optional

HISTORY

Date	Author	Description
2/15/1999	Sadanand Sahasrabudhe	Initial Version

INTRODUCTION

One of the primary goals of the Castor Engine VLDB design was to have a clean abstraction for the SQL generation module. The requirement is to have the SQL generation part independent of the core engine processing functionality. In addition we also want the capability to change the syntax without having to recompile the system.

In this document we talk more about the abstraction and design that is used to generate the syntax so that it can be changed dynamically.

.1 EXAMPLE SQL

A typical SQL statement generated by the SQL engine is shown below.

```
insert into zz_MJ00B00
select a2.CUR_TRN_DT, a2.CUR_TRN_DT,
    a2.MONTH_ID, a2.MONTH_ID,
    a1.REGION_NBR, a1.REGION_NBR,
    sum(a1.TOT_SLS_DLR), WJXBFS1
from REGION_CLASS a1,
    LOOKUP_DAY a2
on a1.CUR_TRN_DT = a2.CUR_TRN_DT
where (a1.CLASS_NBR in (1, 6))
group by a2.CUR_TRN_DT,
    a2.MONTH_ID,
    a1.REGION_NBR
```

The above sample SQL was generated for SQL Server. The same SQL statement when generated for Oracle is shown below.

```
create table zz_MJ00B00 as
select a2.CUR_TRN_DT, a2.CUR_TRN_DT,
    a2.MONTH_ID, a2.MONTH_ID,
    a1.REGION_NBR, a1.REGION_NBR,
    sum(a1.TOT_SLS_DLR) WJXBFS1
from REGION_CLASS a1,
    LOOKUP_DAY a2
where (a1.CUR_TRN_DT = a2.CUR_TRN_DT
and (a1.CLASS_NBR in (1, 6))
group by a2.CUR_TRN_DT,
    a2.MONTH_ID,
    a1.REGION_NBR
```

The two SQL statements shown above, both create implicit tables. Notice the difference between the implicit table creation syntax for the two databases. Also notice the difference between the join syntax for the two databases – SQL server is generating SQL-92 standard syntax whereas Oracle needs the 'traditional' SQL syntax.

Given the following two VLDB syntax generation settings, we can write a syntax generation module that can generate the two SQL statements shown above.

1. JOIN_SYNTAX (with possible values JOIN92 and JOIN89)
2. IMPLICIT_TABLE_SYNTAX (with possible values ORACLE_STYLE and SQLSERVER_STYLE)

.2 FACTORS AFFECTING SQL GENERATION

Section 1.1 gave an introduction on how the engine needs to generate different syntax. We saw how certain 'settings' could be used to control SQL generation. If you take into account the different

situations in which the SQL Engine generates syntax we can come up with many factors that affect the decision to generate syntax at different points. We now discuss the different factors and give examples for each of the factors.

I.2.1 VLDB Setting

As we saw in the example previously, VLDB Settings obviously affect the way syntax gets generated. One setting we saw, JOIN_SYNTAX, governs the way the joins are generated.

I.2.2 Internal Program Setting

Another kind of setting that can affect syntax generation is 'programmatic' setting. By this we mean an internal setting in memory that comes into play during actual execution of the system. The original source or deciding factors for this internal setting may be a combination of the given report instance and multiple VLDB Settings on different objects involved in the report.

A classic example of this kind of setting is 'outer join' setting. Given a report with one or more metrics – the kind of join that needs to be done between different tables is decided by a complex algorithm involving the settings on the different metrics and the report and the support provided by a particular database type for different kinds of outer joins. In the end, when it actually comes to syntax generation all we have is a memory location that indicates two tables have to be 'left outer joined' or 'equi joined' etc.

Now depending on the memory setting we need to generate different syntax. To use the example of types of joins, let us revisit the SQL generated in section 1.1 for SQL Server.

```
insert into ZZTMJ00B00
select a1.CUR_TRN_DT,a2.CUR_TRN_DT,
a2.MONTH_ID,a2.MONTH_ID,
a1.REGION_NBR,a1.REGION_NBR,
sum(a1.TOT_SLS_DLR),WJXBFS1
from REGION_CLASS a1
join LOOKUP_DAY a2
on a1.CUR_TRN_DT = a2.CUR_TRN_DT
where a1.CLASS_NBR in (1, 5)
group by a2.CUR_TRN_DT,
a2.MONTH_ID,
a1.REGION_NBR
```

We observe that the tables *a1* and *a2* are 'equi joined'. If the memory settings had dictated that the tables be 'full outer joined', then the SQL generated is shown below.

```
insert into ZZTMJ00B00
select coalesce (a2.CUR_TRN_DT, a1.CUR_TRN_DT) CUR_TRN_DT,
a2.MONTH_ID,a2.MONTH_ID,
a1.REGION_NBR,a1.REGION_NBR,
sum(a1.TOT_SLS_DLR),WJXBFS1
from REGION_CLASS a1
full outer join LOOKUP_DAY a2
on a1.CUR_TRN_DT = a2.CUR_TRN_DT
where a1.CLASS_NBR in (1, 5)
group by a2.CUR_TRN_DT,
a2.MONTH_ID,
a1.REGION_NBR
```

Notice the changes in the *from* and *select* clauses.

SQL GENERATION AND PATTERNS

In this section we introduce the concept of SQL generation patterns. We will see how patterns are used to generate syntax and how by simply changing a pattern we can generate an entirely different syntax.

A pattern is a string that describes how a given set of argument strings is manipulated to produce a result string.

3. VLDB SYNTAX GRAMMAR

We now introduce the concept of VLDB Syntax Grammar (VSG). VSG is used to describe the syntax that is generated by the SQL Engine. It is a specialized grammar that uses patterns, VLDB settings, memory settings and string arguments to describe the syntax generated.

3.1 EXAMPLE

Let us consider the example of generating the join syntax, specifically we will consider the SQL 92 standard syntax. A typical SQL92 join clause is shown below:

TABLE1 a1
join TABLE2 a2
on (a1. STORE_ID = a2.STORE_ID)
and (a1.ITEM_ID = a2.ITEM_ID)

We could use a context free grammar syntax to describe the above construct.
NODESQL – TableName AliasName
NODESQL – NODESQL JOINTYPE NODESQL JOINSQL

JOINTYPE – *join* | *cross join* | *left outer join* | *full outer join* | *right outer join*

JOINSQL – (COLEXPR)
JOINSQL – JOINSQL *and* JOINSQL

COLEXPR – COLUMN = COLUMN

COLUMN – AliasName.ColumnName

In the above grammar the following variables are resolved to terminating strings that come from different basic SQL entities such as table names, column names, alias names etc.

TableName – obviously the name of the table being joined.
AliasName – name of the alias given to a particular table being joined
ColumnName – name of a taking part in a join

The grammar shown above is intended to describe language syntax. We need a specialized grammar that describes how a language syntax *should be generated* using patterns and settings, hence the concept of a VSG. We now introduce the basic concepts of a VSG.

3.2 VSG – BASIC CONCEPTS

We now introduce the different grammar constructs that make up a VSG. We need to keep in mind that a VSG is a language generation grammar and it is very closely tied to syntax generation paradigms in the Castor SQL Engine.

3.2.1 Applying a pattern

One of the constructs used in a VSG is that of applying a pattern to a set of string arguments to generate a result string. The two elements used in this construct are the pattern that is to be applied and the string arguments that the pattern applies to.

The general form of this construct is shown below.

PATTERN → (string arguments)

A particular example of this construct is shown below.

TABPATTERN → (tablename, tablealias) where *TABPATTERN* is #0 #1.

The pattern that is applied itself can be chosen based on different settings as we will see in the next section.

2.2 Choosing a pattern

Another construct used in a VSG is that of choosing a pattern based on a setting. This setting can be a VLDB Setting or it can be a memory setting. The general form of this construct is shown below.

SETTING(Pattern1 OR Pattern2 OR ... PatternN)

An example of this construct is that of choosing different patterns for joins depending on an internal memory setting for type of joins.

JoinSetting(REGULARJOINPATTERN OR CROSSJOINPATTERN OR LEFTOUTERJOINPATTERN OR FULLOUTERJOINPATTERN)

2.3 Other Constructs

Some of the other constructs used in a VSG are:

1. *vector<XXX>* is a shortcut used to denote a vector of strings.

3 AN EXAMPLE VSG

We will now look at an example VSG. As in section 3.1, we will consider the case of generating SQL92 standard join syntax.

NODESQL – TABPATTERN → (TableName, AliasName)

NODESQL – APPENDPATTERN (JOINSQL ,ONSQL)

JOINSQL – JOINPATTERN → (NODESQL, NODELSQL)

JOINPATTERN – InternalJoinSetting (REGULARJOINPATTERN OR CROSSJOINPATTERN OR LEFTOUTERJOINPATTERN OR FULLOUTERJOINPATTERN OR RIGHTOUTERJOINPATTERN)

ONSQL – ONPATTERN → ((vector<COLEXPR>), (vector<COLEXPR>))

COLEXPR – COLPATTERN → (AliasName, ColumnName)

TABPATTERN – #0 #1

ONPATTERN – #0 = #1 | #0#< and #* #>

REGULARJOINPATTERN – #0 join #1

APPENDPATTERN – #0#< #*#>

CROSSJOINPATTERN – #0 cross join #1

LEFTOUTERJOINPATTERN -- #0 left outer join #1

FULLOUTERJOINPATTERN – #0 full outer join #1

RIGHTOUTERJOINPATTERN – #0 right outer join #1

4. CASTOR SYNTAX GENERATION

In the previous sections we looked at patterns and how they are used in VSGs. We now discuss how this all ties together in providing the flexibility and database independence we need in Castor.

4.1 VSG ANALYSIS

As we saw earlier a VSG describes how the Castor SQL Engine should generate syntax. The real power and flexibility of the VSG comes from the fact that it uses patterns to drive syntax generation. If these patterns are changed the syntax generated changes accordingly. The patterns themselves are not hard coded inside the SQL Engine. They exist as properties on a DBMS object. By changing the pattern properties on the database object we can dynamically change the syntax generated without the need to recompile the engine.

Thus any part of the SQL syntax that is generated by a rule of the general form *PATTERN* → *(string arguments)* can be modified dynamically by changing the pattern.

In addition to the effect of changing patterns, we also have the constructs where patterns are selected depending on VLDB settings. Thus when we change a VLDB setting, it causes a different pattern to be selected and changes the SQL generation.

The one construct, which does not have a truly dynamic effect is that of choosing patterns based on internal memory settings. The internal memory settings are set by an algorithm based on settings on objects and internal logic and hence can only be indirectly controlled.

'atterns and VSG Grammars : Create Table

his document describes the patterns and VSG grammars that were used in the Create Table part of the yntax Abstraction. The purpose of this syntax abstraction is to increase the flexibility of the Castor Engine ith regard to the SQL generation part. The flexibility is increased by removing the hard encoding from ie code. This is replaced by patterns, which can be changed when necessary without the need to recompile ie code again. As an example, this could eliminate the problems when a database vendor changes the way does joins.

) General Patterns

his section gives a few examples of patterns, which are of a general nature and are thus suitable for not ily the Create Table VSG but also for other VSGs.

PPENDPATTERN #0#<#*#>
This pattern concatenates all its arguments without any separators.

MPTYPATTERN
This pattern returns an empty string.

EPARATORAPPENDPATTERN #1#<#0#*#>
This pattern needs at least two arguments and uses its first argument as a separator in the concatenation of the rest of its arguments.

PPLYPATTERN #0|#1
This pattern uses the result of its first argument as input for its second.

) The Create Table VSG

his section focuses on the specific VSG for the Create Table syntax. This VSG is based on the 'DDL and ML syntax' document. The VSG in this section only focuses on the generation of the QLStatementBlock. The starting symbol of this VSG is CREATETABLEPATTERN.

REATETABLEPATTERN	CreateTableSetting(EXPLICIT OR IMPLICITORACLE OR IMPLICITSQLSERVER OR IMPLICITINFORMIX)
ABLEPRESTATPATTERN	SEPARATORAPPENDPATTERN→ (gSPACE, TablePreStatement1, TablePreStatement2)
ABLEPOSTSTATPATTERN	SEPARATORAPPENDPATTERN→ (gSPACE, TablePostStatement1, TablePostStatement2)
REATETABLEPATTERN	create #0 table #1 #2#3
REATEDATAPATTERN	(SEPARATORAPPENDPATTERN→ (gSPACE, coln, data_type, Constraint)
EMPTABLESPACEPATTERN	APPENDPATTERN→ (TempTableSpace)
ICPIDPATTERN	RICPIDSetting(RI_Constraint OR Partition_Index_definition)
ABLEMIDSTATPATTERN	SEPARATORAPPENDPATTERN (gSPACE, TableMidStatement1, TableMidStatement2)
NSERTINTOPATTERN	insert into #0 #1
ELECTHINTPATTERN	SEPARATORAPPENDPATTERN→ (gEOL, SelectHintStatement, FromStatement, WhereStatement, GroupByStatement, HavingStatement)
SSELECTHINTPATTERN	APPENDPATTERN→ (gAS, gSPACE, SELECTHINTPATTERN)
NTOPATTERN	into #0 #1 #2#3
ELECTONLYPATTERN	SelectHintStatement
XPLICIT	SEPARATORAPPENDPATTERN→ (gEOL, TABLEPRESTATPATTERN, CREATETABLEPATTERN, CREATEDATAPATTERN,

	TEMPTABLESPACEPATTERN, RICPIDPATTERN, TABLEMIDSTATPATTERN, INSERTINTOPATTERN, SELECTHINTPATTERN, TABLEPOSTSTATPATTERN)
] MPLICITORACLE	SEPARATORAPPENDPATTERN→ (gEOL, TABLEPRESTATPATTERN, CREATETABLEPATTERN, TEMPTABLESPACEPATTERN, APPENDPATTERN(gAS, SELECTHINTPATTERN), TABLEPOSTSTATPATTERN)
] MPLICITSQLSERVER	SEPARATORAPPENDPATTERN→ (gEOL, TABLEPRESTATPATTERN, SELECTONLYPATTERN, INTOPATTERN, TEMPTABLESPACEPATTERN, FromStatement, WhereStatement, GroupByStatement, HavingStatement, TABLEPOSTSTATPATTERN)
] MPLICITINFORMIX	SEPARATORAPPENDPATTERN→ (gEOL, TABLEPRESTATPATTERN, SELECTHINTPATTERN, INTOPATTERN, TEMPTABLESPACEPATTERN, TABLEPOSTSTATPATTERN)

The VSG described above uses a few constants with the following meaning:

gEOL	End Of Line character
gSPACE	A single space
gAS	The string 'as'

Insert start

B

CreateTable VSG

ABSTRACT

The CreateTable VSG presented in this document is intended to increase the flexibility in the generation of Create Table syntax. The patterns described in this document have been replaced by a single pattern, which is described in the VSG patterns and Usage document. Please refer to that document.

HISTORY

Date	Author	Description
4/20/99	Leon Bun	Initial Version
4/28/99	Leon Bun	ColumnLine clarification
5/7/99	Leon Bun	Addition of section 2 + rewrite of section 3
5/17/99	Leon Bun	Addition of section 4 : Indexes & Teradata/UDB CreateTable VSG modification
6/3/99	Leon Bun	Rewrote section 3.3 & 3.4. Moved 4.2 into 3.4. Moved differences from high level to lowest possible.
6/16/99	Leon Bun	Modified 4.1 & 3.4 to conform with the implemented code. Added section 5 with implementation details.
10/14/99	Leon Bun	Changed abstract. Document is obsolete see VSG Patterns and Usage for up to date information.

1. INTRODUCTION

VSG Grammar presented in this document is based on the code found in SyntaxSQL::GenCreateStmt. The next section presents the VSG itself and a small example to clarify it. This section gives some general patterns that will be frequently used (hopefully).

AppendPat : #0#<#*#>

The following three operators are used in this document:

- PATTERN→(string arguments)
- SETTING (Pattern1 OR Pattern2 OR ... PatternN)

The PATTERN and SETTING-operator are explained in the document 'Zen and the Art of VLDB Syntax Generation'.

Some pattern notations and their meanings:

- #< begin repeat block
- #> end repeat block
- #0 first argument also called explicit variable
- #1 second argument also called explicit variable
- #* remaining arguments also called implicit variable
- | pipe, use output of left part as input for right
- Non-Terminal words appearing on the left and possibly on the right
- Terminal words appearing only on the right

2. NEW VSG ARGUMENTS AND OPERATORS

This section examines the new VSG operators and the interactions between arguments and patterns. First, subsection 2.1 examines the interaction between patterns and the arguments to which they are applied. Second, subsection 2.2 introduces two new VSG operators, head and tail.

2.1 OF STRINGS, VECTORS, AND PATTERNS

The construction of the columnlines in the CreateTable VSG is the most complex part. This complexity is caused by the inclusion of both strings and string-vectors. The patterns that are used to define a VSG are input independent. This means, that the meaning of a pattern does not change based on the type of its inputs. The presence of a vector in the input can, however, result in an implicit loop, which becomes explicit in the code.

In order to determine if the output of a pattern is a string or a vector of strings the following simple rules are used:

1) If an explicit variable is linked with a vector argument, then the output of the pattern is a vector of strings.
2) If no explicit variables are linked with vector arguments, then the output of the pattern is a string.

In the rules above an explicit variable is indicated by the #2 token, which points to the third argument. The only implicit variable is represented by the #* token, which points to the arguments that remain after all the explicit variables have been linked.

For example, the #0#<#*#>#1 pattern has two explicit variables (#0 and #1), and one implicit variable (#*). Depending on its arguments this pattern can produce either a string or a vector of strings. In the following, some arguments are used to clarify this.

- Input: vector<LastName>, vector<FirstName>, USA, vector<CityData>
- Output: vector of strings, with each string contains a LastName followed by 'USA', the entire CityData vector and finally a FirstName.
- Input: LastName, FirstName, vector<PersonalData>
- Output: string, which contain LastName followed by the contents of the entire PersonalData vector and FirstName at the end.
- Input: LastName, FirstName, ZipCode, Address
- Output: string, consisting of LastName, ZipCode, Address, and FirstName

Note that when a pattern is applied it always needs at least as many arguments as it has explicit variables.

A final note on the difference between vectors that are linked to explicit variables and those linked to the implicit variable. When a vector is linked to the implicit variable, it is treated as a number of unrelated strings. In this case, it is not possible to see the difference between using that vector as the input argument or its string elements as separate input arguments.
When a vector is linked to an explicit variable, the pattern to which it contains is executed for each of the elements of that vector separately. This results in a loop and a vector of string results. A consequence of this approach is that when multiple vectors are linked to explicit variables, these vectors have to have the same length. If this is not the case, then the pattern can not be applied as it has not enough arguments.

2.2 OF HEADS AND TAILS.

When a vector of strings has to be used to produce a string of its elements with a certain separator in between, then the rules above do not allow it. In order to still follow the simple rules, two additional operators on vectors are needed. They are the head and the tail operator. Their names suggests exactly what they are meant to do. The head operator takes a non-empty vector as input and produces the first string element of that vector. The tail operator also takes a non-empty vector as input and removes the first element of the input vector, which results in a vector output.

For example given a vector of lines the following pattern can be used to separate those lines with commas, which results in a string.

- SeparatorPattern→(head(vector<lines>), tail(vector<lines>))
- SeparatorPattern : #0#<,#*#>

On the other hand if no head or tail operator is used, it is not possible to get the desired result without bending the rules or inventing new operator.

- Sep2Pattern→(vector<lines>)
- Sep2Pattern : #<,#*#>

In this case, the first comma has to be deleted by a new operator.

- Sep3Pattern→(vector<lines>)
- Sep3Pattern : #0#<,#*#>

In this case, the result is a vector equal to the input vector, because according to rule the vector gets linked to an explicit variable. In order to get the desired result, the rule has to be extended to allow a vector to be linked to multiple variables. This could get confusing and is much better handled by the introduction of the head and tail operator as described above.

3. THE CREATE TABLE VSG

This section presents the Create Table VSG. This presentation has been split into four parts. First, the patterns dealing with greater chunks of SQL are presented in subsection 3.1. Subsection 3.2 presents the details of the first line of the CreateTable syntax, the so-called CreateLine. Next, subsection 3.3 describes how the syntax needed for columns in CreateTable syntax is generated. Finally, subsection 3.4 presents the details of the so-called PostColumnPart.

3.1 OVERVIEW PATTERNS

TotalCreateTable : AppendPat → (CreateLine, ColumnPart, PostColumnPart)
CreateLine : AppendPat → (CreatePart, QualPart, TablePart, DescrPart, TTNamePart, OptionPart)
PostColumnPart : AppendPat→(SpacePart, PostColumnIndexPart)

The overview patterns give an overview of the patterns generated to produce a create table statement. TotalCreateTable is the starting symbol, which shows the three main parts of the create table statement. The most important part is generated by ColumnPart, which combines the column names, types and constraints. CreateLine takes care of the correct syntax before the columns start. The PostColumnPart takes care of the syntax after the columns have been enumerated.

3.2 DETAIL PATTERNS OF CREATELINE

CreatePart : AppendPat→(gCREATE, gSPACE)
QualPart : AppendPat→(Qualifier, gSPACE)
TablePart : AppendPat→(gTABLE, gSPACE)
DescrPart : AppendPat →(Descriptor, gSPACE)
TTNamePart : AppendPat→(TempTableName, gSPACE)
OptionPart : AppendPat→(Option, gSPACE)

The CreateLine consists of a number of parts, which are either always present (such as CreatePart, TablePart, and TTNamePart) or which might be absent (such as QualPart, DescrPart, and OptionPart). An optimization in the case of possible absent parts is to only include them when they contain something useful. This can be realized by a test in the code. The advantage of this approach is that no extra (harmless) characters clutter the formatting of the SQL syntax.

3.3 DETAILS OF COLUMNPART

ColumnLine : VecAppendPat→(gTABKEY, (vector<ColName>), (vector<ColType>), (vector<ColConstraint>))
VecAppendPat : #0#1#0#2#0#3
ColumnPart : ControlInsertPat→(Control, head(vector<ColumnLine>), tail(vector<ColumnLine>))
ControlInsertPat : (#0#1#<,#0#*#>)

The syntax of the columns are generated by ColumnLine with the help of the VecAppendPat pattern, which results in a vector of ColumnLines. These ColumnLines are subsequently used in the generation of the ColumnPart. This adds the control to each line, the parentheses at the beginning and end, and the comma's between the ColumnLines. The inclusion of the commas between the ColumnLines is the cause of the use of the head and tail operator.

3.4 DETAIL PATTERNS OF POSTCOLUMNPART

The TotalCreateTab always appends the SpacePart and PostColumnPartition to the end of the Create Table syntax. Both these parts are optional and the remarks regarding optional syntax in subsection 3.2 also apply in this case. They can also be used to include database specific syntax.

SpacePart : AppendPat→(gSPACE, Space, gSPACE, TableSpaceStatement)
PostColumnPartition : AppendPat→(Control, PPIPart)

The terminal 'TableSpaceStatement' is optional and can be used to specify the TableSpace in which a particular table has to be created.

PPIPart : PPISetting(gEMPTY, PartitionKeyColPart, PrimaryIndexPart)

The PPISetting is a setting based on the value of the Intermediate Table Index setting. The results of the PPISetting are summarized in the following table, which also takes the index creation into account.

Intermediate Table Index	DBMS	Partitioning Key	Create Index
0	All	No	No
1	Teradata	Yes	No
1	DB2/UDB	Yes	Yes
1	Other	No	No
2	Teradata or DB2/UDB	No	No
2	Other	No	Yes

Note that the only part of PartitionKeyColPart & PrimaryIndexPart that has to be filled in are the column names. This means it makes no difference which pattern is used, as they both can be applied on the same arguments. The rest of those two patterns only consist of keywords. The creation of create index SQL is discussed in section 4.1.

PartitionKeyColPart : AppendPat(PartitionPart, KeyPart, IndexColPart)
PartitionPart : AppendPat(gPARTITIONING gSPACE)
KeyPart : AppendPat(gKEY, gSPACE)

The three lines above present the details of the UDB partitioning key generation. The most interesting par t is the IndexColPart, which takes care of the generation of the column names, which are used in the partitioning key. Subsection 4.1 describes the exact definition of IndexColPart.

PrimaryIndexPart : AppendPat(PrePrimaryPart, IndexColPart, PostPrimaryPart)
PrePrimaryPart : AppendPat(PrimaryPart, IndexPart)
PrimaryPart : AppendPat(gPRIMARY, gSPACE)

The lines above present the details of the primary index generation for Teradata. Again the most interesting part is the generation of the column names by IndexColPart described in section 4.1. The other non-terminals take care of the generation of the necessary keywords. The IndexPart non-terminal is described in detail in section 4.1.

4. INDEX CREATION AND PARTITIONING KEYS

This section explains the VSG for the creation of indexes on tables. Subsection 4.1 presents the VSG, which is needed in order to create indexes.

4.1 CREATE INDEX VSG

This section describes the VSG grammar needed in the generation of Create Index syntax.

```
IndexCreation      : AppendPat(PreIndexPart, IndexColPart, PostIndexPart)
PreIndexPart       : AppendPat(CreateIndexPart, IndexOnPart)
IndexColPart       : ColumnCommaAppendPat(head(vector<KeyColName>),
                     tail(vector<KeyColName>))
PostIndexPart      : AppendPat(gSPACE, IndexPostStatement)
```

IndexCreation consists of three parts. The PreIndexPart generates the SQL before the index key. The IndexColPart generates the key upon which the index will be based. The index key is influenced by the 'Max Columns in Index' setting. This setting determines the maximum number of columns, which could be included in an index or partition key. Finally the PostIndexPart takes care of the IndexPostStatement. The value for this IndexPostStatement is taken directly from the 'Index Post String' setting.

```
ColumnCommaAppendPat    : (#0#<,#*#>)
```

The most interesting pattern of this section is the ColumnCommaAppendPat, which is used in the generation of key, upon which the index is based. It start with a left parenthesis followed by each of the columnnames, which are if necessary separated by comma's, and finishes with a right parenthesis.

```
CreateIndexPart    : AppendPat(CreatePart, IndexPart)
IndexPart          : AppendPat(gINDEX, gSPACE)
IndexOnPart        : AppendPat(IndexNamePart, OnPart, TTNamePart)
IndexNamePart      : AppendPat(gI_, TTNamePart)
OnPart             : AppendPat(gON, gSPACE)
```

The five lines above present the details of the PreIndexPart. All they do is use the AppendPat pattern to append various parts. The CreatePart and TTNamePart can be found in section 3.2.

5. VSG IMPLEMENTATION DETAILS

This section gives a short overview of the implementation of Create Table VSG and Create Index VSG. Section 5.1 presents the details of the Create Table VSG implementation. The Create Index VSG implementation is described in section 5.2. Section 5.2.1 describes CreateIndexColumnsVSG, an auxilary method.

5.1 CREATE TABLE VSG

In this section the CreateStmtVSG method, which implements the Create Table VSG, is discussed. The function of this method is to generate create table SQL. It is called from the Parse method in the DFCCompUnit class. It generates the column names, types, and possible constraints based on the input value of the attribute and fact column names and types and on the constraints input vector. The VLDBInfo is used to get the tablesettings, that is the qualifier, descriptor, etc. The VLDBInfo is also used to get the Intermediate Table Index setting. This setting is used in the case of Teradata and DB2 databases, which need to generate a partitioning key.

Checks are made to ensure correct input. This entails an equal number of attribute column names and types. The same goes for the fact column names and types. Furthermore, no create table SQL can be generated if not at least one column is present. Note that a table can exist with only fact columns.

5.2 CREATE INDEX VSG

This section discusses the CreateIndexStmtVSG method, which takes care of the implementation of the Create Index VSG. The function of this method is to generate create index SQL. It is called from the Parse method in the DFCCompUnit class. Using the names of the attribute columns, the VLDBInfo, and the temp table name the create index statement is generated and returned to the caller. Note that a check is performed on the size of the attribute column names vector, as an index can not be generated without any attribute column names.

5.2.1 CreateIndexColumnsVSG

This section discusses the CreateIndexColumnsVSG method. The function of this method is to get the names of the attributes, which will be used in the index creation by CreateIndexStmtVSG. This depends on the Max_Col_In_Index setting, which is read from the VLDBInfo. This method is called by CreateIndexStmtVSG and by CreateStmtVSG. It assumes that it receives a non-empty vector of attribute names. Note that this is not checked.